ISWIM-LIKE, INTERACTIVE McG360

B.M. Leavenworth McG360 PROGRAMMERS GUIDE. IBM Thomas J. Watson Research Center, Yorktown Heights, New York report no. RC 2693, 35 pages. 1969 November 7.

McG360 is an interactive programming language currently running under TSS/360 and CP67/CMS. It was developed by W.H. Burge and is similar in philosophy to the Iswim language of Landin and to a derivative language implemented at MIT. McG360 has a simple syntax and semantics, and is written in Lisp 1.5. A "message" in McG is either an expression to be evaluated, or a definition to be stored in the environment for future reference. Messages are translated into an internal tree representation which is then executed by an interpreter.

This guide is meant to serve as a combined reference manual and introduction to the language. An appendix gives examples of the use of McG360 at the terminal. (Introduction)

REPRESENTING ALGORITHMS

Robert M. Shapiro and Harry Saint. THE REPRESENTATION OF ALGORITHMS. Rome Air Development Center (Griffiss Air Force Base, New York) report no. RADC-TR-69-313, volume II, 94 pages. 1969 September.

The problem of representing mathematical processes is considered in the context of digital computer software and hardware. (Abstract)

Procedures are outlined which make possible the translation of a sequentially defined algorithm into a powerful representation of highly concurrent execution of the algorithm wherein each operation takes place when (1) the necessary operand values are available, (2) enough decisions have been made to guarantee that the operation will be required, and (3) enough decisions have been made to guarantee that no logically prior claim can be made on the algorithmic parts incolved. All sequencing is stripped out except that which is given by data dependencies or by priorities for part use. In the process, control has been dismembered and the useful information which it carries has been broken down into individual ordering relations. (From the paper)

contributions

THE CWIC/360 SYSTEM, A COMPILER FOR WRITING AND IMPLEMENTING COMPILERS

Erwin Book, Dewey Val Shorre, and Steven J. Sherman (System Development Corporation, 2500 Colorado Avenue, Santa Monica, California 90406. 1970 April 14

ABSTRACT

Cwic/360 (Compiler for Writing and Implementing Compilers) is a meta-compiler system. It is composed of compilers for three special-purpose languages, each intended to permit the description of certain aspects of translation in a straightforward, natural manner. The Syntax language is used to describe the recognition of source text and the construction from it of an intermediate tree structure. The Generator language is used to describe the transformation of the tree into appropriate object language. The MOL/360 language is used to provide an interface with the machine and its operating system.

This paper describes each of these languages, presents examples of their use, and discusses the philosophy underlying their design and implementation.

INTRODUCTION

In order to enlist the aid of a computer in the solution of a problem, one must find a way of expressing the problem in such a way as to be able to communicate with and take advantage of the machine. In the "good old days," one learned the language of the computer and translated an algorithm into that language. It was then necessary to check out the translation before proceeding to check out the algorithm.

Eventually, it was discovered that programs could be written to accomplish this translation, making possible the expression of the algorithm in a language expressly designed for that purpose. Such a "higher-level" programming language is far more natural to human users than machine language, and the time needed to check out translation was reduced considerably. In effect, a high-level language translator turns a computer into a special-purpose pseudo-machine, whose order code is the set of all operations and instructions implemented in the translator's source language.

However, the translating programs--called compilers and/or interpreters--were extremely expensive to build. Consequently, language designers attempted to create languages which were useful over as broad a range of applications as possible so that the cost could be spread over the widest possible market. This led to inefficiencies, for the following reasons. First, the translators became even bigger and more expensive to build. Second, the languages could not be completely right for any given problem, because they were applicable to so many different problems. They were "jacks of all trades and masters of none."

The advent of metacompilers or compiler-compilers changes this situation radically. A metacompiler or compiler-compiler is a program whose input is a description of the syntax of a programming language, plus a description of its semantics, i.e., of its equivalent in terms of a machine language. The

language in which such a description is coded is called a metalanguage. The output of a metacompiler is a compiler for the described language. This compiler can then be used to process a program written in that language: i.e., to examine it for syntactic correctness and to translate it into a machine language or its equivalent.

A metacompiler assists the task of compiler-building by automating its non-creative aspects, those aspects that are the same regardless of the language which the produced compiler is to translate. This makes possible the design of languages which are appropriate to the specification of a particular problem. It reduces the cost of producing processors for such languages to a point where it becomes economically feasible to begin the solution of a problem with language design.

The CWIC/360 compiler-compiler system is a tool used both in the design of programming languages and the construction of the concomitant compilers, interpreters, or translators.

OVERVIEW

In its most general form a metacompiler is a program, written for a Machine M, which will accept specifications for a programming language L_j; and its equivalent in the language of Machine M_i, and produce a compiler which runs on Machine M. Source programs which are the input to this compiler are written in Language L_j. The output of this compiler is object language which runs on Machine M_i.

Figure I shows the operation of a compiler that processes programs written in Language L_j into an intermediate tree structure and then processes this tree into object language for Machine M_i. This compiler runs on Machine M. The primitive routines that interface with Machine M and its operating system are the support package.

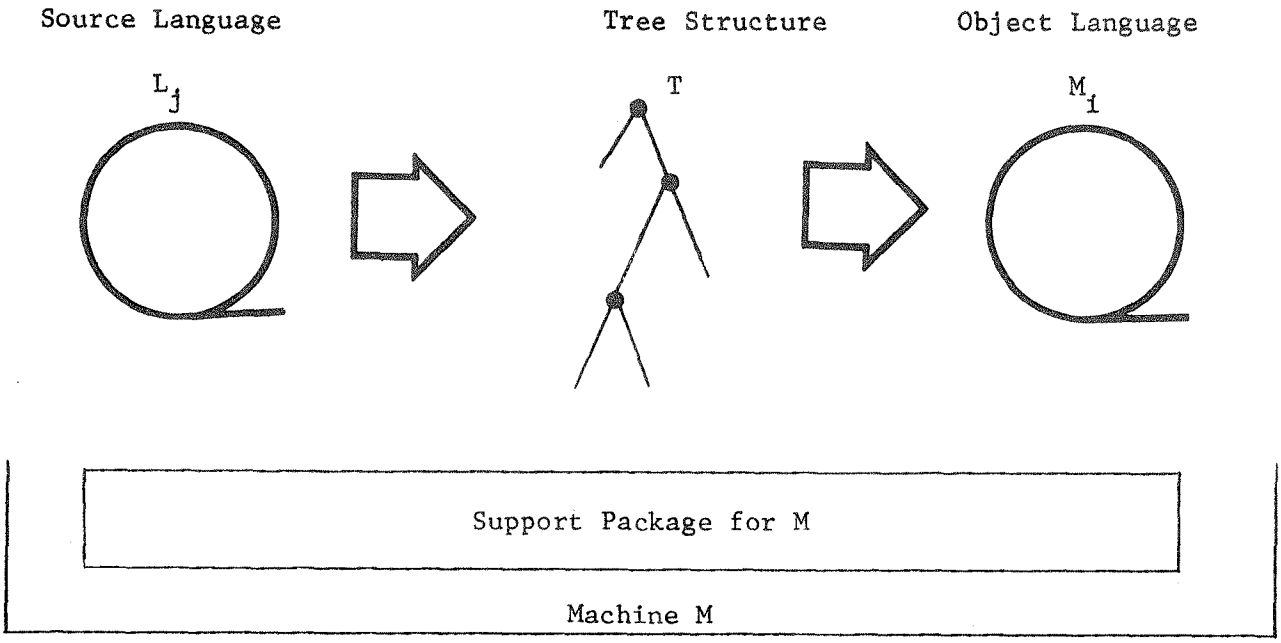


Figure I

Figure II shows the coding of the compiler in Figure I. The first transformation, namely source language to tree structure, is coded in the SYNTAX language of CWIC. The second transformation, to wit, tree structure to object language, is coded in the GENERATOR language. The support package is coded in an MOL (Machine-Oriented Language) for Machine M.

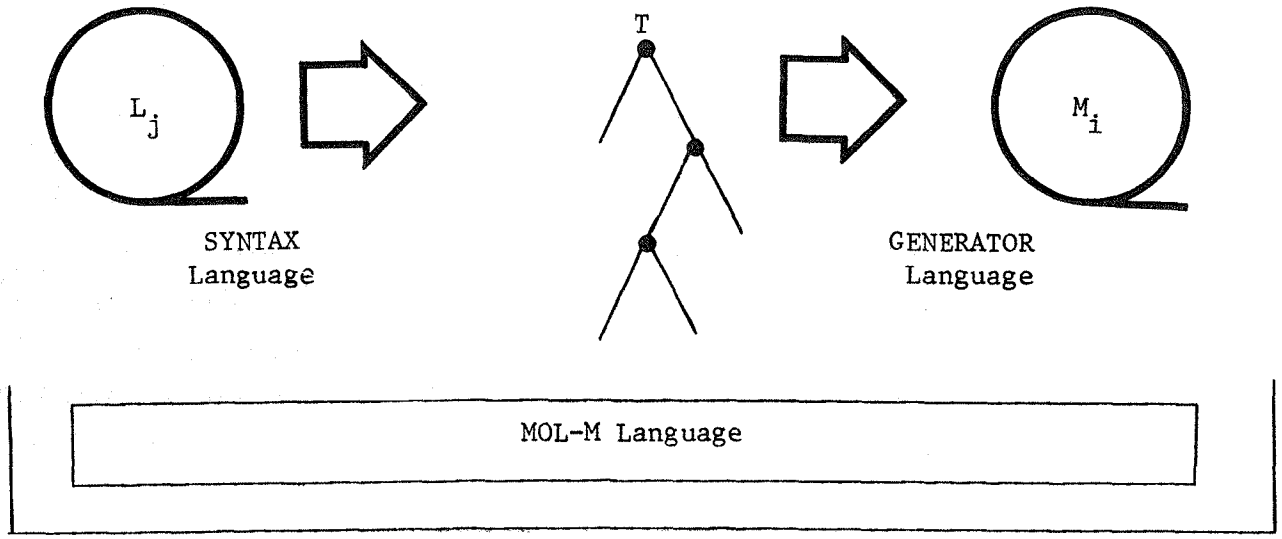


Figure II

The CWIC/360 system consists of three languages and their associated compilers, plus a support package of MOL-360 routines:

1. A SYNTAX language for the syntactic description of a programming language;
2. A GENERATOR language for the semantic description of a programming language;
3. An MOL-360 (Machine-Oriented Language) for the primitives and machine-dependent aspects of the compiler.

An object compiler is written in three sections, each section in a different language.

THE SYNTAX LANGUAGE

The SYNTAX language is used to code a program that transforms the strings of characters of a source language into a tree structure representation, thereby defining the grammar rules for the source language and mapping it into a set of instructions for a pseudo-machine. This SYNTAX language is a version of BNF (Backus-Naur Form) with extensions for tree building. A top-to-bottom, non-deterministic algorithm is used. The program coded in the SYNTAX language is compiled into running machine code rather than a set of tables for an interpreter.

The SYNTAX language is designed to simplify the following aspects of compiler construction:

1. Scanning;
2. Parsing (precedence of operators);
3. Lexicographic analysis;
4. Hashing and construction of dictionary or symbol table entries;
5. Detection of errors;
6. Pictorial reporting of errors;
7. Recovery from errors to continue compilation;
8. The construction of an intermediate parse tree.

THE GENERATOR LANGUAGE

The GENERATOR language is used to code a program that transforms into an object language the tree formed by a program in the SYNTAX language or by other GENERATOR routines. The object language may be machine code or assembly code, in which case a compiler has been produced. The object language may be strings of characters of another higher level language, in which case a translator has been produced. The tree may be used as a set of instructions for immediate execution, in which case an interpreter has been produced. The GENERATOR language is a powerful programming language with features that simplify such problems of compiler construction as the following:

1. Optimization of code sequences by delineation of special cases;
2. Writing global optimization routines over the directed graph of a program;
3. Evaluation of constant expressions at compile time;
4. Elimination of common subexpressions;
5. Mixed mode and fixed point arithmetic;
6. Detection of duplicate and undefined labels;
7. Planting of data and instructions of object programs in core;
8. Features for writing and reading tree structures and dictionary on disc or tape, thereby facilitating the construction of multi-pass compilers;
9. Creation of dictionaries which reflect programs with unlimited block structure;
10. Storing of symbolic or numeric information into the dictionary;
11. Interrogation of the dictionary.

The GENERATOR language borrows from LISP its data structures, functional orientation, and automatic storage allocation capability. Its syntax is more

convenient, however, reflecting its design as specifically applicable to compiler building.

THE MOL-360 LANGUAGE

The compiler being constructed runs on a particular computer, namely the IBM 360, under some operating system (OS/360). It needs a set of routines called a SUPPORT package which performs a number of machine-oriented tasks for the compiler. Specifically, the SUPPORT package must:

1. Establish an interface between the compiler and the machine, expressing many of the compiler's basic operations in the machine's terms.
2. Establish an interface between the compiler and the operating system, including such matters as input/output and code conversions.
3. Define the data structures of the compiler in terms of the storage structures chosen on a machine to represent them (this is also done by programs written in the GENERATOR language which may be used to compile open code and to plant binary data in memory).

Obviously, the SUPPORT package is highly machine-oriented, and it could be written in assembly language. But, since assembly-language coding is very tedious, and since the design of a special-purpose language to replace it is greatly facilitated by the CWIC system itself, the SUPPORT package is coded in a machine-oriented language (MOL-360). An MOL is a substitute for assembly language.. It has a compiler's grammar and an assembler's vocabulary; that is, the form of the language is Algol-like, but the user has direct control of the machine's memory and registers, just as he would in assembly language.

THE CWIC/360 SYSTEM

The CWIC/360 system consists of three compilers, plus a set of MOL-360 routines that have been compiled by the CWIC MOL-360 compiler into machine code, with its

associated dictionary. Each CWIC compiler accepts, in addition to its source-language programs, any machine code and associated dictionary that have been produced by any CWIC compiler.

The CWIC MOL-360 compiler accepts MOL-360 language and in one pass produces 360 machine code plus an associated dictionary.

The CWIC/360 SYNTAX compiler accepts SYNTAX language and in one pass produces 360 machine code plus an associated dictionary.

The CWIC/360 GENERATOR compiler is a two-pass compiler. The first pass accepts GENERATOR language and produces a tree on disc or tape. The second pass accepts the tree and produces 360 machine code plus an associated dictionary.

CWIC/360 operates on any IBM 360 with an H core or larger. It operates under the PCP, MFT, or MVT versions of OS/360 and is completely compatible with OS. It is completely self-contained, using only the supervisor and data management package of OS.

The CWIC/360 system has its own load program that is used to load any program or compiler produced with CWIC/360, including CWIC/360, into core and execute it. This load program serves the function of the OS linkage editor but is only 700 bytes long. The total system occupies 15 2311 disc cylinders or one quarter that number of 2314 cylinders. If 2311 discs are used the track overflow feature must be available.

Each of the compilers of CWIC/360 is defined in CWIC/360; that is to say, they have been written in the SYNTAX, GENERATOR, and MOL-360 languages and compiled through CWIC/360. Since the CWIC/360 system produces itself or other compilers in machine code and defines itself, it is completely extensible. It is far more extensible than the so-called extensible programming systems that compile extensions into a basic high-order programming language and cannot express changes that are not inherent in the operators and data types of the basic language.

COMPILATION OF A COMPILER UNDER CWIC/360

The object compiler is coded in the SYNTAX and GENERATOR languages. If part of the compiler cannot be expressed in these languages for one reason or another, additional MOL routines must be coded.

The additional MOL routines, if necessary, are compiled by the CWIC MOL-360 compiler into the SUPPORT package, forming an augmented SUPPORT package and dictionary.

The SYNTAX language portion of the object compiler is compiled by the CWIC SYNTAX compiler into the augmented support package and dictionary.

The GENERATOR language portion of the object compiler is compiled by the CWIC GENERATOR compiler into the result of the preceding operation. This forms the complete object compiler and its dictionary, ready for execution. This process is illustrated in Figure III.

Three important consequences of this approach should be made clear.

1. Partial recompilation and incremental compilation may be made into an already compiled compiler. New routines may be added and old routines overridden.
2. It is possible to write parts of a single program in "n" different languages and have the resultant programs communicate and operate as a unit. Each language used is designed to be right for part of the total job.
3. The use of many compilers to produce a single program is possible under CWIC.

A SIMPLE INTERPRETER IN CWIC/360

As a sample application of the CWIC/360 system, this section presents a simple interpreter, coded in the SYNTAX and GENERATOR languages. This interpreter permits the declaration of variables and their preset values, and the use of these variables (and constants) in computations via an assignment statement.

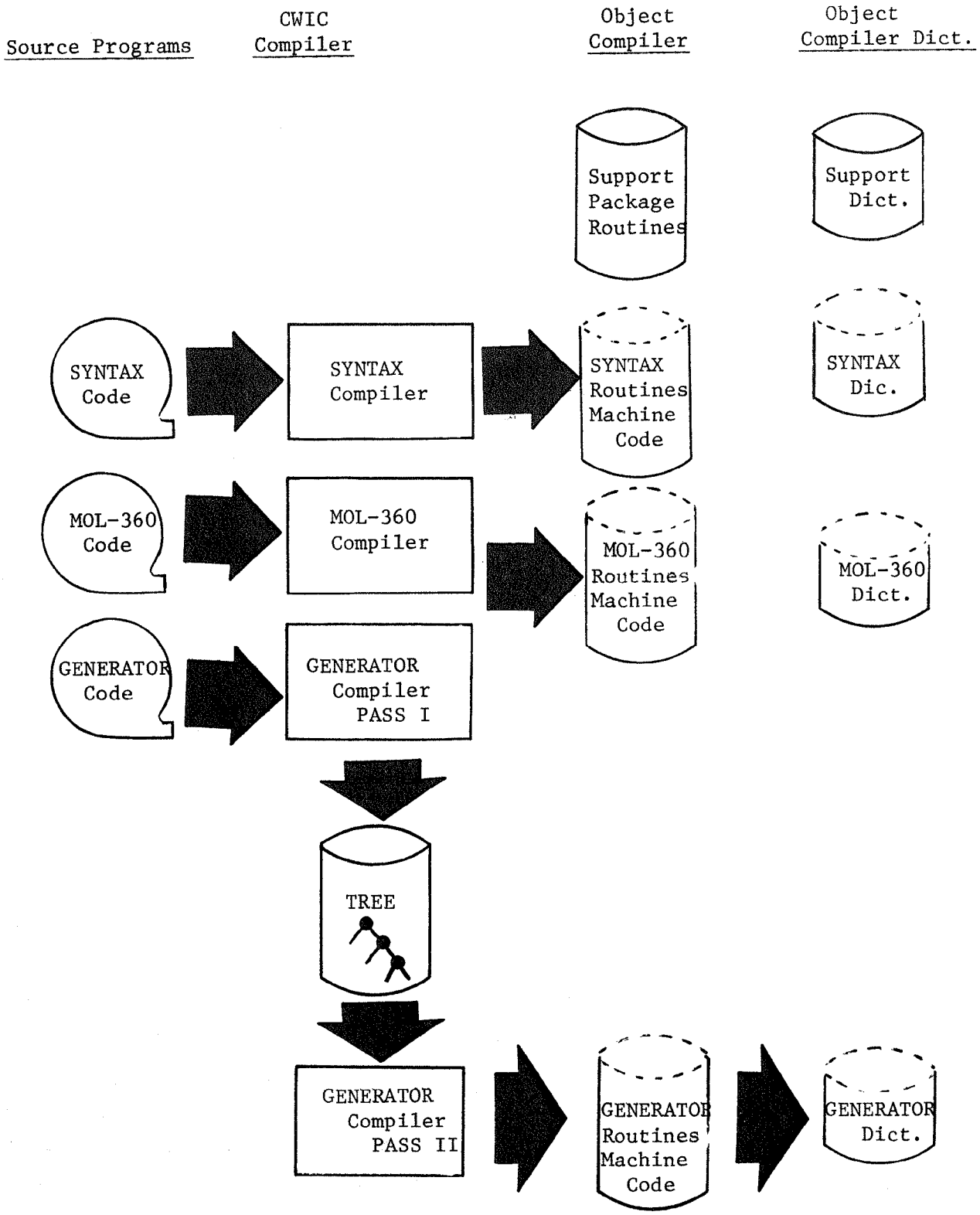

Source Programs
CWIC Compiler
Object Compiler
Object Compiler Dict.
Support Package Routines
Support Dict.
SYNTAX Code
SYNTAX Compiler
SYNTAX Routines Machine Code
SYNTAX Dic.
MOL-360 Code
MOL-360 Compiler
MOL-360 Routines Machine Code
MOL-360 Dict.
GENERATOR Code
GENERATOR Compiler PASS I
TREE
GENERATOR Compiler PASS II
GENERATOR Routines Machine Code
GENERATOR Dict.

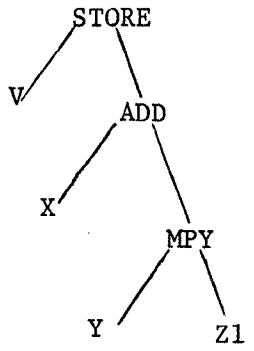
Figure III

Figure IV shows the syntactic specifications of the language. The syntax equations LET, DGT, and ALPHNUM define character classes. The equation ID defines an identifier as a letter followed by zero or more alphanumerics. The equation NUM defines a number as a digit followed by zero or more digits.

A program is defined as a sequence of declarations and statements, followed by END. A declaration is distinguished by the operator =, as opposed to the := used in statements.

A declaration recognizes an identifier and the value to which it is preset, and passes this information to the GENERATOR language routine DECL.

```
X = 5; Y = 12; Z1 = 5;
V := X + Y * Z1;
.END
```




```
.SYNTAX
PROGRAM = $(ST | DECLARATION) '.END' ;
DECLARATION  = ID '=' NUM ';' :EQU!2 DECL[*1];
ST = ID ':=' EXP ';' :STORE!2 COMPILE[ *1];
EXP = TERM $('+' TERM :ADD!2);
TERM = FACTOR $('*' FACTOR :MPY!2);
FACTOR = ID / '(' EXP ')' / NUM;
LET:  'A'/ 'B'/ 'C'/ 'D'/ 'E'/ 'F'/ 'G'/ 'H'/ 'I'/ 'J'/ 'K'/ 'L'/ 'M'/
      'N'/ 'O'/ 'P'/ 'Q'/ 'R'/ 'S'/ 'T'/ 'U'/ 'V'/ 'W'/ 'X'/ 'Y'/ 'Z';
DGT:  '0'/ '1'/ '2'/ '3'/ '4'/ '5'/ '6'/ '7'/ '8'/ '9';
ALPHNUM:  LET / DGT;
ID .. LET $ALPHNUM;
NUM .. DGT $DGT MAKENUMBER[];
.FINISH
.STOP SETUP PROGRAM
```

Figure IV

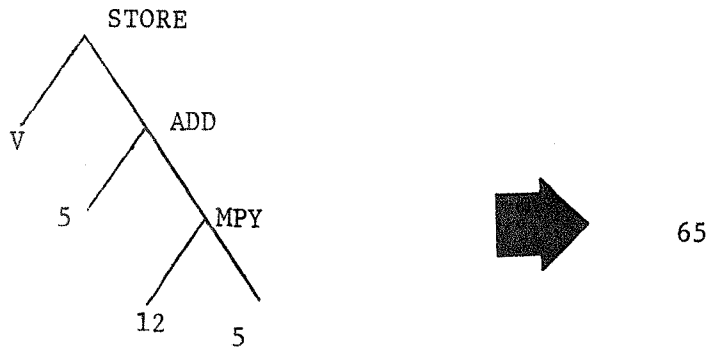
A statement recognizes an identifier and the expression whose value is to be computed and assigned to it. The expression is defined in a manner akin to that used in BNF except that iteration is used rather than left recursion. The operations recognized are addition (+) and multiplication (*), with the latter taking precedence unless overridden by the use of parentheses. Identifiers and numbers may also be expressions.

The syntax rules also contain directions as to the building of the intermediate parse tree. For example, in the definition of term, the code specifies that a pair of factors is to be grouped as the branches of a tree with a MPY node at the root.

As each assignment statement is parsed, its constituent identifier and expression are gathered at a STORE node, and this information is passed to the GENERATOR language routine COMPILE. Figure IV shows an example of both the source language and a tree for a typical statement, and a trace is shown in the Appendix.

Figure V shows the GENERATOR language portions of the interpreter. The generator DECL simply sets the DEF attribute of its first parameter (some identifier) to the value of its second parameter (some number). The generator COMPILE sets the DEF attribute of its first parameter (some identifier) to the result of evaluating the expression which is the second parameter. This evaluation is performed by the generator EVAL.

EVAL also illustrates the pattern-matching features of the GENERATOR language. For example, if the input to EVAL is an identifier (as determined by IDP), its definition will be returned. If it is a number (as determined by NUMBERP), then the number itself will be returned. In the third instance, the program employs a convenient shorthand feature of the language, specifying that the input may be an expression attached to either an ADD or a MPY node (#V), and returns either the sum or the product (#U) or the expressions at the node's branches, depending upon which node was recognized. A trace is shown in the Appendix.




```
.GENERATOR
DECL(EQU[X, Y]) => DEF:(X) := Y
COMPILE(STORE[X, Y]) => DEF:(X) := EVAL(Y); PRINT(DEF:(X))
EVAL(IDP(X)) => DEF:(X)
     (NUMBERP(X)) => X
     (#V1[EVAL(X), EVAL(Y)]) => #U1
          #V = ADD, MPY
          #U = X + Y, X * Y
.FINISH
.STOP SETUP PROGRAM
```

Figure V

This interpreter was coded in the space of about fifteen minutes, and was executed successfully on the second attempt. One shudders to think of the time that would have been required to perform the same task had the interpreter been coded in a higher level language of the JOVIAL or PL/1 class, much less in assembler language.

DEBUGGING UNDER CWIC

The three primary tools of debugging are partial recompilation, the ability to print out symbolic information, and the debugging statements and declarations.

The debugging facilities are:

1. The Syntax error message

   A printout of the input line with an asterisk under the point of error.

2. The Backtrace

A printout of the name of the routine where the error was detected, with a symbolic print of its parameters and the name of the routine that called it. Also, its local variables are printed if the routine is a generator. The same information is printed about the routine that called the routine that called the routine that detected the error, and so on up to the top level routine. This is followed by the program's global variables and however much dictionary has been constructed to that point.

3. The MOL-360 TRACE and UNTRACE statements

```
.TRACE ST, EXP, TERM, FACTOR, ID, NUM, DECL, EVAL;
.UNTRACE TERM, FACTOR;
```

When a TRACE statement is executed all subsequent entries and exits from the routine being traced are monitored and printouts occur. Printouts vary depending upon the language in which the routine being traced was originally written.

For syntax rules the input line, the current scan position as an asterisk under the input line, and the parse tree up to the point of entry are printed. Upon leaving a syntax rule the same information is printed, showing the changes which occur through the action of the routine. VALUE = TRUE or VALUE = FALSE is also printed, as appropriate.

For generator routines the parameters are printed upon entry and the parameters and local variables are printed upon exit. Also, the value being returned by the generator is shown in the place of its first parameter.

For MOL-360 routines the parameters are printed upon entry as well as exit.

The UNTRACE statement turns off the trace. Since these statements are executed dynamically, conditional or loop traces may be programmed, to wit,

```
.IF A < 64
   .THEN TRACE MARY; .END
```

An actual trace, made during execution of the simple interpreter, is shown in the Appendix.

4. The MOL-360 RENAME declaration

```
.RENAME SAM .AS MARY;
```

The compiled routine SAM is renamed MARY. A new definition of SAM should be coded that does some special debugging (e.g., printing the value of global variables). Then a call on MARY will result in the same action that the original SAM performed. This is followed by perhaps more special action before the RETURN.

```
SAM(X):
   PRINT(FELIX); MARY(X); PRINT(SLIM);
   .RETURN
```

All previously compiled calls on SAM will now call this new version of SAM, which calls the old SAM, now called MARY.

ADVANTAGES OF USING CWIC

The advantage of using CWIC to build compilers is that it reduces the cost of building compilers drastically, for the following reasons.

1. The compiler description, written in the CWIC language, is probably 1/5 to 1/4 the size that it would be if it were written in an ordinary high-level programming language. This means fewer people are needed or less time will be taken in producing the compiler.

2. The compiler will be easy to read and modify.

3. The CWIC system constructs compilers automatically using checked-out modules covering most aspects of compilation.

4. The CWIC system builds error detection facilities into compilers which will help debug it.

5. Using CWIC to build compilers, one can produce better code for object programs than if the compiler was coded by hand. This is due to the fact that optimization algorithms can be expressed in CWIC more easily than in a language not so suited to such algorithms. This clearly enables the production of better object programs for a fixed amount of programmer and/or machine time dollars.

APPLICATIONS

The CWIC system is suited to the design and implementation of compilers or interpreters covering a wide range of applications:

1. Conventional programming languages such as FORTRAN, JOVIAL, SPL, ALGOL, or even LISP.

2. Special-purpose programming languages designed to fit a limited field such as the actuarial or statistical.

3. Programming languages which express symbol manipulation, such as algebraic simplification, symbolic differentiation, or game playing.

4. Applications in which the output is not a computer program but rather a set of instructions which drive a machine or printed reports for human consumption.

5. The production of tables which drive operating systems.

6. The design and implementation of compilers for machine-oriented languages, languages for system programmers, and given machines, which fall in the void between assembly language and so-called higher level languages like PL/1; these languages have a compiler-language grammar and an assembly-language vocabulary.

APPENDIX

A Trace of the Execution of the Simple Interpreter

```
ENTER_   ID          EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                 *
       STAR =.NIL
LEAVE_   ID          EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                 *
       STAR =(X)
       VALUE=TRUE
ENTER_   NUM         EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                   *
       STAR =(X)
LEAVE_   NUM         EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                    *
       STAR =(5 X)
       VALUE=TRUE
ENTER_   DECL        EXIT DECLARAT
       CCCC0005        (EQU X 5)
LEAVE_   DECL        EXIT DECLARAT
       FFFFBFFB        5
       00003309        EQU
       000030E7        X
       FFFFBFFB        5
ENTER_   ID          EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                     *
       STAR =.NIL
LEAVE_   ID          EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                     *
       STAR =(Y)
       VALUE=TRUE
ENTER_   NUM         EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                       *
       STAR =(Y)
LEAVE_   NUM         EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                       *
       STAR =(12 Y)
       VALUE=TRUE
ENTER_   DECL        EXIT DECLARAT
       CCCC000B        (EQU Y 12)
LEAVE_   DECL        EXIT DECLARAT
       FFFFBFF4        12
       00003309        EQU
       000030E8        Y
       FFFFBFF4        12
ENTER_   ID          EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                         *
       STAR =.NIL
LEAVE_   ID          EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                         *
       STAR =(Z1)
       VALUE=TRUE
ENTER_   NUM         EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                           *
       STAR =(Z1)
LEAVE_   NUM         EXIT DECLARAT
       INPUT=X = 5 ; Y = 12; Z1 = 5;
                           *
       STAR =(5 Z1)
       VALUE=TRUE
```

```
ENTER_   DECL      EXIT DECLARAT
         00000012    (EQU Z1 5)
LEAVE_   DECL      EXIT DECLARAT
         FFFFBFFB    5
         00003309    EQU
         0000330C    Z1
         FFFFBFFB    5
ENTER_   ID        EXIT DECLARAT
         INPUT= V := X + Y*Z1;
                  *
         STAR =.NIL
LEAVE_   ID        EXIT DECLARAT
         INPUT= V := X + Y*Z1;
                   *
         STAR =(V)
         VALUE=TRUE
ENTER_   ST        EXIT PROGRAM
         INPUT= V := X + Y*Z1;
                  *
         STAR =.NIL
ENTER_   ID        EXIT ST
         INPUT= V := X + Y*Z1;
                  *
         STAR =.NIL
LEAVE_   ID        EXIT ST
         INPUT= V := X + Y*Z1;
                   *
         STAR =(V)
         VALUE=TRUE
ENTER_   EXP       EXIT ST
         INPUT= V := X + Y*Z1;
                     *
         STAR =(V)
ENTER_   TERM      EXIT EXP
         INPUT= V := X + Y*Z1;
                     *
         STAR =(V)
ENTER_   FACTOR    EXIT TERM
         INPUT= V := X + Y*Z1;
                     *
         STAR =(V)
ENTER_   ID        EXIT FACTOR
         INPUT= V := X + Y*Z1;
                     *
         STAR =(V)
LEAVE_   ID        EXIT FACTOR
         INPUT= V := X + Y*Z1;
                      *
         STAR =(X V)
         VALUE=TRUE
LEAVE_   FACTOR    EXIT TERM
         INPUT= V := X + Y*Z1;
                     *
         STAR =(X V)
         VALUE=TRUE
LEAVE_   TERM      EXIT EXP
         INPUT= V := X + Y*Z1;
                     *
         STAR =(X V)
         VALUE=TRUE
ENTER_   TERM      EXIT EXP
         INPUT= V := X + Y*Z1;
                      *
         STAR =(X V)
ENTER_   FACTOR    EXIT TERM
         INPUT= V := X + Y*Z1;
                      *
         STAR =(X V)
ENTER_   ID        EXIT FACTOR
         INPUT= V := X + Y*Z1;
                      *
         STAR =(X V)
LEAVE_   ID        EXIT FACTOR
         INPUT= V := X + Y*Z1;
                       *
         STAR =(Y X V)
         VALUE=TRUE
LEAVE_   FACTOR    EXIT TERM
         INPUT= V := X + Y*Z1;
                       *
         STAR =(Y X V)
         VALUE=TRUE
ENTER_   FACTOR    EXIT TERM
         INPUT= V := X + Y*Z1;
                       *
         STAR =(Y X V)
ENTER_   ID        EXIT FACTOR
         INPUT= V := X + Y*Z1;
                       *
         STAR =(Y X V)
LEAVE_   ID        EXIT FACTOR
         INPUT= V := X + Y*Z1;
                        *
         STAR =(Z1 Y X V)
         VALUE=TRUE
```

```
LEAVE_   FACTOR    EXIT TERM
       INPUT= V := X + Y*Z1;
                         *
       STAR =(Z1 Y X V)
       VALUE=TRUE
LEAVE_   TERM      EXIT EXP
       INPUT= V := X + Y*Z1;
                       *
       STAR =((MPY Y Z1) X V)
       VALUE=TRUE
LEAVE_   EXP       EXIT ST
       INPUT= V := X + Y*Z1;
                     *
       STAR =((ADD X (MPY Y Z1)) V)
       VALUE=TRUE
ENTER_   EVAL      EXIT COMPILE
       0000001F      (ADD X (MPY Y Z1))
ENTER_   EVAL      EXIT EVAL
       000030E7      X
LEAVE_   EVAL      EXIT EVAL
       FFFFBFFB      5
       FFFFFFFD      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFC      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFD      .UNDEF
ENTER_   EVAL      EXIT EVAL
       0000001B      (MPY Y Z1)
ENTER_   EVAL      EXIT EVAL
       000C3CE8      Y
LEAVE_   EVAL      EXIT EVAL
       FFFFBFF4      12
       FFFFFFFD      .UNDEF
       FFFFFFFC      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFD      .UNDEF
ENTER_   EVAL      EXIT EVAL
       0000330D      Z1
LEAVE_   EVAL      EXIT EVAL
       FFFFBFFB      5
       FFFFFFFC      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFC      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFD      .UNDEF
       FFFFFFFC      .UNDEF
LEAVE_   EVAL      EXIT EVAL
       FFFFBFC4      60
       0000330C      MPY
       000030E8      Y
       0000330D      Z1
       FFFFBFF4      12
       FFFFBFFB      5
       00000028      (1)
LEAVE_   EVAL      EXIT COMPILE
       FFFFBFBF      65
       0000330B      ADD
       000030E7      X
       0000001B      (MPY Y Z1)
       FFFFBFFB      5
       FFFFBFC4      60
       (65) <------- THE ANSWER
       00000026      (0)
LEAVE_   ST        EXIT PROGRAM
       INPUT=.END
             *
       STAR =.NIL
       VALUE=TRUE
```